ES 204 – Numerical Methods in Engineering

2nd Semester AY 2017-2018

Machine Problem 02

Submitted by:

Eldion Vincent H. Bartolo

2010-28167

Master of Science in Electrical Engineering

University of the Philippines - Diliman

Estimation of the Developed Power of Hydraulic Impulse Turbine through Newton's Interpolating Polynomial and Cubic Splines.

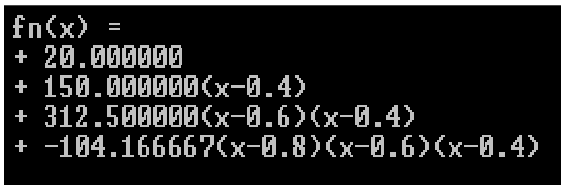
Problem Statement

The power developed by a hydraulic impulse turbine with a certain penstock diameter is to be estimated through the mathematical models created from Newton's Interpolating Polynomial of order 3 (NIPO3) and Cubic Splines (CS).

Results and Discussion

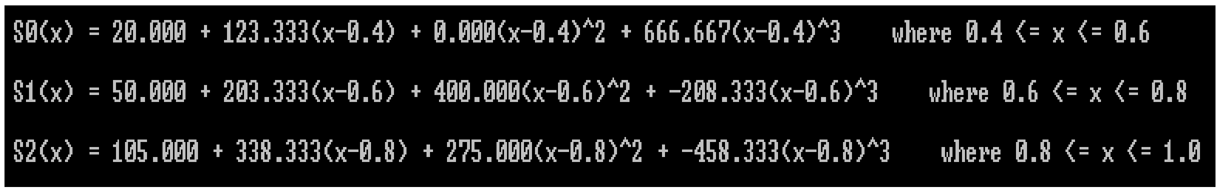
C Programs (*see Appendix*) are created to develop the models for NIPO3 and CS.

Figure 1 shows the model generated using NIPO3, while the model for CS is shown in figure 2.



```
fn(x) =
+ 20.000000
+ 150.000000(x-0.4)
+ 312.500000(x-0.6)(x-0.4)
+ -104.166667(x-0.8)(x-0.6)(x-0.4)
```

Figure 1: Model created through Newton's Interpolating Polynomial of order 3



```
S0(x) = 20.000 + 123.333(x-0.4) + 0.000(x-0.4)^2 + 666.667(x-0.4)^3    where 0.4 <= x <= 0.6

S1(x) = 50.000 + 203.333(x-0.6) + 400.000(x-0.6)^2 + -208.333(x-0.6)^3    where 0.6 <= x <= 0.8

S2(x) = 105.000 + 338.333(x-0.8) + 275.000(x-0.8)^2 + -458.333(x-0.8)^3    where 0.8 <= x <= 1.0
```

Figure 2: Model created through Cubic Splines

A superimposed plot of NIPO3 and CS is shown in fig. 3. As seen from the fig. 3, CS has greater estimates for the value of the power compared to NIPO3 at diameters: 0.4< D <0.6 and

0.8< D <1.0. On the other hand NIPO3 has larger estimates compared to CS at 0.6< D <0.8.

The mathematical models intersect at the given data points (e.g. D = 0.4, 0.6, 0.8, and 1.0). The differences can be attributed to the different modelling of the two algorithms. CS has different models for each sub-interval but NIPO3 has the same model throughout the subintervals.

Given a penstock diameter of 0.9 m, CS will give an estimated power of **141.1250 MW** while NIPO3 will give a value of **140.3125 MW**.

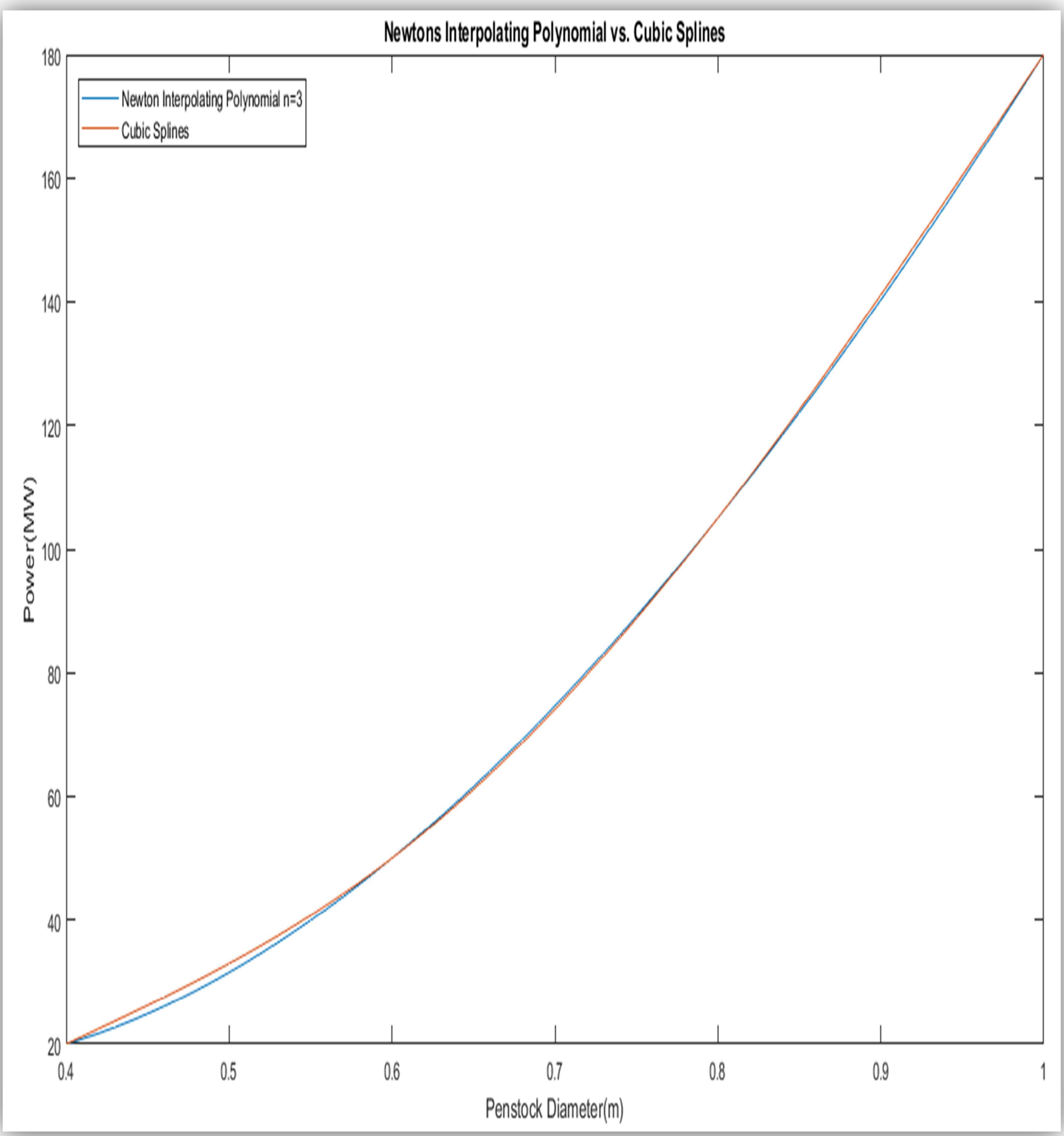


Figure 3: Plots of the models created from NIPO3 and CS

Problems encountered

The generation of the code for NIPO3 is quite easy. On the other hand the code for CS is troublesome because the creation of the tridiagonal matrix is very tricky. Aside from this, CS needs an algorithm for finding the solutions of a system of linear equations.

Thus I used LU Decomposition-Crout's Method for solving the value of ci's.

Another problem encountered is the use of pointers. Pointers are used to create a more neat C code for CS since there are a lot of operations needed for solving the coefficients of the CS. I accomplished these challenges through further reading.

References

- S. C. Chapra, R. P. Canale – Numerical Method for Engineers, 7th edition.
- K. J. Yap, ES 204 Notes – Curve Fitting: Collocation and Interpolation
- https://www.tutorialspoint.com/cprogramming/c_pointers.htm

Appendix

See next pages for the source code of Newton's Interpolating Polynomial order 3 and Cubic Splines.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*
    ES 204 MP2 Problem #1a - Newton's Interpolating Polynomial order 3
*/

double evaluate(double a[],double xi[],double x ){
    int i,j;
    double fx = 0;
    double prod;

    for(i=0;i<4;i++){
        prod = a[i];
        if(i>0){
            for(j=i-1;j>=0;j--){
                prod = prod*(x-xi[j]);
            }
        }
        fx = fx + prod;
    }
    return fx;
}

int main()
{
    double xi[4] = {0.4,0.6,0.8,1};
    double fxi[4] = {20,50,105,180};
    double a[4];
    double derivative1[3], derivative2[2], derivative3;
    double res,num;
    int i,j,k ;

    for(i=0;i<3;i++){
        derivative1[i] = (fxi[i+1] - fxi[i])/(xi[i+1] - xi[i]);
    }
    for(i=0;i<2;i++){
        derivative2[i] = (derivative1[i+1] - derivative1[i])/(xi[i+2] - xi[i]);
    }
    derivative3 = (derivative2[1] - derivative2[0])/(xi[3] - xi[0]);

    a[0] = fxi[0];
    a[1] = derivative1[0];
    a[2] = derivative2[0];
    a[3] = derivative3;

    printf("\n fn(x) = \n");
    for(i=0;i<4;i++){
        printf(" + ");
        printf("%0.6lf",a[i]);
        if(i>0){

            for(j=i-1;j>=0;j--){
                printf("(");
                printf("x-%0.1lf",xi[j]);
                printf(")");
            }
        }
        printf("\n");
    }
    num = 0.9;
    res = evaluate(a,xi,num);
    printf("\n Value at %0.1lf m = %0.9lf MW\n",num,res);

    return 0;
}

```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*
    ES 204 MP2 Problem #1b - Cubic Splines with Natural End Conditions
*/

//Computes the value of System of linear equations through LU Decomposition
double *LUDecomp(double a[4][4],double b[4]){
    double U[4][4], L[4][4];
    static double y[4], x[4];
    int total, col, row, k, i, j;

    col = sizeof(a[0])/sizeof(a[0][0]);
    row = col;

    //Initialize the L and U Matrix to zero
    for (j=0; j<col; j++){
        for (i=0; i <row; i++){
            U[i][j] = 0;
            L[i][j] = 0;
        }
    }

    for (j=0; j<col; j++){
        U[j][j] = 1;
        for (i=0; i <row; i++){
            // Compute for elements of L
            if (i>=j){
                L[i][j] = a[i][j];
                for(k = 0; k < j; k ++){
                    L[i][j] = L[i][j] - L[i][k]*U[k][j];
                }
            }
        }
        for (i=0; i <row; i++){
            // Compute for elements of U
            if (i>j) {
                U[j][i] = a[j][i];
                for(k = 0; k<j; k++){
                    U[j][i] = U[j][i] - L[j][k]*U[k][i];
                }
                U[j][i] = U[j][i]/L[j][j];
            }
        }
    }

    for(i=0;i<row;i++){
        //Forward Substitution
        y[i] = b[i];
        for(k = 0; k < i; k ++) {
            y[i] = y[i] - L[i][k]*y[k];
        }
        y[i] = y[i]/L[i][i];
    }
    for(i=(row-1); i >= 0;i--){
        //Backward Substitution
        x[i] = y[i];
        for(k=(row-1); k > i; k --) {
            x[i] = x[i] - U[i][k]*x[k];
        }
    }

    return x;

}

int main()
{
    double x[4] = {0.4,0.6,0.8,1};
    double f[4] = {20,50,105,180};
    double z[4][4];
    double c[4], u[4];
    double h[3],a[3],b[3],d[3];
    double divdiff[2];
    double *p;
    double res, num;
    long i, j;


    //Compute for Values of hi
    for(i=0;i<3;i++){
        h[i] = x[i+1] - x[i];
    }
    //Compute for Values of ui
    for(i=0;i<4;i++){
        if (i==0 || i==3){
            u[i] = 0; // condition for Natural Ends
        }else{
            divdiff[1] = (f[i+1]-f[i])/h[i];
            divdiff[0] = (f[i]-f[i-1])/h[i-1];
            u[i] = 3*(divdiff[1] - divdiff[0]);
        }
    }

    // Initialize to zero the z matrix
    for(i=0;i<4;i++){
        for(j=0;j<4;j++){
            z[i][j] = 0;
        }
    }

    //Compute values for z matrix
    for(i=0;i<4;i++){
        for(j=0;j<4;j++){
            if(i==j){
                //Middle Diagonal Elements
                if(i==0 || i==3){
                    z[i][i] = 1;
                }else{
                    z[i][i] = 2*(h[i-1]+h[i]);
                }
            }else if(i>0 && i <3){
                if(i == j-1){
                //Upper diagonal elements
                    z[i][j] = h[i];
                }else if(i==j+1){
                //Lower diagonal elements
                    z[i][j] = h[i-1];
                }
            }
        }
    }

    // Get the value of ci's through LU Decomposition
    p = LUDecomp(z,u);
    for(i=0;i<4;i++){
        c[i] = *(p+i);
    }

    printf("\n The cubic splines equations are: \n\n");
    for(i=0;i<3;i++){
        // Compute for ai,bi,di and print results
        a[i] = f[i];
        b[i] = ( (f[i+1]-f[i])/h[i] ) - (h[i]/3)*(2*c[i]+c[i+1]);
        d[i] = (c[i+1]-c[i])/(3*h[i]);

        printf(" S%d(x) = ",i);
        printf("%0.3lf",a[i]);
        printf(" + %0.3lf(x-%0.1lf)",b[i],x[i]);
        printf(" + %0.3lf(x-%0.1lf)^2",c[i],x[i]);
        printf(" + %0.3lf(x-%0.1lf)^3",d[i],x[i]);
        printf("    where %0.1lf <= x <= %0.1lf",x[i],x[i+1]);
        printf("\n\n");
    }

    //Evaluate D = 0.9
    num = 0.9;
    res = a[2]+b[2]*(num-x[2]) + c[2]*pow(num-x[2],2)+d[2]*pow(num-x[2],3);
    printf(" Value at %0.1lf m = %0.9lf MW\n",num,res);

    free(p);
    return 0;
}
```

Integration of the Temperature of a Continuous Stirred Tank Reactor using Adaptive Quadrature with Simpson's 1/3 Rule

Problem Statement

The temperature of a Continuous Stirred Tank Reactor (CSTR) is to integrated form 0 to 4 minutes through Adaptive Quadrature with Simpson's 1/3 Rule.

Results and Discussion

The model for the temperature of CSTR is given by fig. 4.

$$T(t) = 20 + 10\left(1 - e^{-2t}\right)\sin\left(e^{t} - 1\right)$$

Figure 4: Mathematical model for the Temperature of CSTR.

A C Program (*see Appendix*) is created to implement Adaptive Quadrature. The adaptive quadrature is implemented by the creating a recursive function that keeps calling itself while the tolerances for each subinterval is not met or the maximum limit for the number of sub intervals is met. The number of subintervals created is also monitored through the use of pointers.

The created program will yield an integral equal to **84.5020.** The algorithm converged with **206** sub-intervals.

Problems encountered

The generation of code is troublesome because there are multiple operations needed for algorithm (e.g. I-fullstep, I-halfstep). Thus the source code is divided into multiple functions. The implementation of adaptive quadrature is also hard to imagine since it keeps on dividing its subintervals until the tolerances are within limits. Thus I searched for varieties of ways to implement adaptive quadrature. One that caught my interest is the use of recursive functions and this was used in my program.

References

- S. C. Chapra, R. P. Canale – Numerical Method for Engineers, 7th edition.
- K. J. Yap, ES 204 Notes – Numerical Integration Part 1 and 2.
- https://www.math.wustl.edu/~victor/classes/ma449/s09.txt
- https://www.programiz.com/c-programming/c-recursion
- https://www.tutorialspoint.com/cprogramming/c_recursion.htm
- https://www.tutorialspoint.com/cprogramming/c_pointers.htm

Appendix

Please see next pages for the source code of Adaptive Quadrature with Simpson's 1/3 Rule.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define maxerr 0.00001

/*
    ES 204 MP2 Problem #2 - Adaptive Quadrature with Simpson's 1/3 Rule
*/

double fn(double t){
    double temp;

    temp = 20+10*(1-exp(-2*t))*sin(exp(t)-1);

    return temp;
}
double compFull(double h,double xi, double xii){

    return (h/6)*(fn(xi)+4*fn(xi+(h/2)) + fn(xii));
}
double compHalf(double h,double xi, double xii){

    double hlf;

    hlf = fn(xi)+4*fn(xi+(h/4)) + 2*fn(xi+(h/2));
    hlf = hlf + 4*fn(xi+((3*h)/4)) + fn(xii);
    hlf = (h/12)*hlf;

    return hlf;
}
double adaptivequad(double a0, double b0, double a, double b,double maxsubint, double
*subint){
    double h,xi,xii;
    double Ifull,Ihalf,err,tol,mean,Iapp;

    Iapp = 0;
    h = b-a;
    xi = a;
    xii = b;

    Ifull = compFull(h,xi,xii);
    Ihalf = compHalf(h,xi,xii);

    err = fabs(Ifull - Ihalf)/15;
    tol = ((h)/(b0-a0))*(maxerr);

    if(err<=tol){
        Iapp = Iapp + Ihalf;
        //printf("Interval [%0.5lf,%0.5lf], error: %0.7lf\n",xi,xii,err);
    }else if(*subint >=maxsubint){
        printf("The maximum subinterval is exceeded\n");
        Iapp = Iapp + Ihalf;
        return Iapp;
    }else{
        mean = (a+b)/2;
        //Call again the recursive adaptive quadrature
        (*subint) = (*subint) + 1;
        Iapp = Iapp + adaptivequad(a0,b0,a,mean,maxsubint,subint); // Left Sub Interval
        Iapp = Iapp + adaptivequad(a0,b0,mean,b,maxsubint,subint); // Right Sub interval
    }

    return Iapp;

}

int main()
{
    double a,ai,b,bi,h,xi,xii;
    double Ifull,Ihalf,err,tol,mean,Iapp;
    int converge;
    double maxsub, subinter;

    subinter = 1;
    maxsub = pow(2,10);
    Iapp = adaptivequad(0,4,0,4,maxsub,&subinter);

    printf("\n\n Answer = %0.9lf\n",Iapp);
    printf(" Number of sub intervals = %0.0lf\n",subinter);
    return 0;
}
```